

Apache Struts 2 Tutorial

Tutorialspoint.com

Struts2 is popular and mature web application framework based on the MVC design pattern. Struts2 is not just the next version of Struts 1, but it is a complete rewrite of the Struts architecture.

The WebWork framework started off with Struts framework as the basis and its goal was to offer an enhanced and improved framework built on Struts **to make web development easier for the developers**.

After some time, the Webwork framework and the Struts community joined hands to create the famous Struts2 framework.

The framework is designed to streamline the full development cycle, from building, to deploying, to maintaining applications over time. Apache Struts 2 was originally known as WebWork 2. For more detail kindly check tutorialspoint.com/struts_2

Struts 2 framework features:

Here are some of the great features that may force you to consider Struts2:

1. **POJO forms and POJO actions** - Struts2 has done away with the Action Forms that were an integral part of the Struts framework. With Struts2, you can use any POJO to receive the form input. Similarly, you can now see any POJO as an Action class.
2. **Tag support** - Struts2 has improved the form tags and the new tags allow the developers to write less code.
3. **AJAX support** - Struts2 has recognised the take over by Web2.0 technologies, and has integrated AJAX support into the product by creating AJAX tags, that function very similar to the standard Struts2 tags.
4. **Easy Integration** - Integration with other frameworks like Spring, Tiles and SiteMesh is now easier with a variety of integration available with Struts2.
5. **Template Support** - Support for generating views using templates.
6. **Plugin Support** - The core Struts2 behaviour can be enhanced and augmented by the use of plugins. A number of plugins are available for Struts2.
7. **Profiling** - Struts2 offers integrated profiling to debug and profile the application. In addition to this, Struts also offers integrated debugging with the help of built in debugging tools.
8. **Easy to modify tags** - Tag markups in Struts2 can be tweaked using Freemarker templates. This does not require JSP or java knowledge. Basic HTML, XML and CSS knowledge is enough to modify the tags.
9. **Promote less configuration** - Struts2 promotes less configuration with the help of using default values for various settings. You don't have to configure something unless it deviates from the default settings set by Struts2.

Struts 2 Environment Setup:

Our first task is to get a minimal Struts 2 application running. This chapter will guide you on how to prepare a development environment to start your work with Struts 2. I assume that you already have JDK (5+), Tomcat and Eclipse installed on your machine. If you do not have these components installed then follow the given steps on fast track:

Step 1 - Setup Java Development Kit (JDK):

You can download the latest version of SDK from Oracle's Java site: Java SE Downloads. You will find instructions for installing JDK in downloaded files, follow the given instructions to install and configure the setup. Finally set PATH and JAVA_HOME environment variables to refer to the



Tutorials Point, Simply Easy Learning

directory that contains java and javac, typically java_install_dir/bin and java_install_dir respectively.

If you are running Windows and installed the SDK in C:\jdk1.5.0_20, you would have to put the following line in your C:\autoexec.bat file.

```
set PATH=C:\jdk1.5.0_20\bin;%PATH%
set JAVA_HOME=C:\jdk1.5.0_20
```

Alternatively, on Windows NT/2000/XP, you could also right-click on My Computer, select Properties, then Advanced, then Environment Variables. Then, you would update the PATH value and press the OK button.

On Unix (Solaris, Linux, etc.), if the SDK is installed in /usr/local/jdk1.5.0_20 and you use the C shell, you would put the following into your .cshrc file.

```
setenv PATH /usr/local/jdk1.5.0_20/bin:$PATH
setenv JAVA_HOME /usr/local/jdk1.5.0_20
```

Alternatively, if you use an Integrated Development Environment (IDE) like Borland JBuilder, Eclipse, IntelliJ IDEA, or Sun ONE Studio, compile and run a simple program to confirm that the IDE knows where you installed Java, otherwise do proper setup as given document of the IDE.

Step 2 - Setup Apache Tomcat:

You can download the latest version of Tomcat from <http://tomcat.apache.org/>. Once you downloaded the installation, unpack the binary distribution into a convenient location. For example in C:\apache-tomcat-6.0.33 on windows, or /usr/local/apache-tomcat-6.0.33 on Linux/Unix and create CATALINA_HOME environment variable pointing to these locations.

Tomcat can be started by executing the following commands on windows machine, or you can simply double click on startup.bat

```
%CATALINA_HOME%\bin\startup.bat

or

C:\apache-tomcat-6.0.33\bin\startup.bat
```

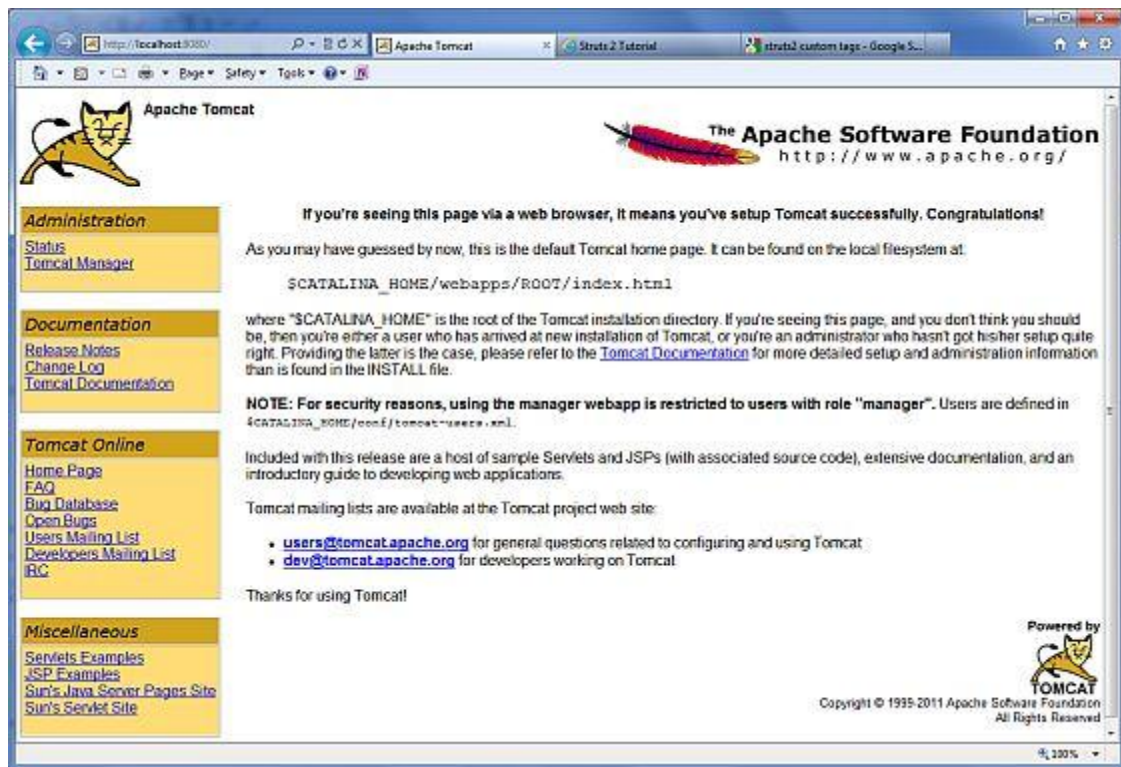
Tomcat can be started by executing the following commands on Unix (Solaris, Linux, etc.) machine:

```
$CATALINA_HOME/bin/startup.sh

or

/usr/local/apache-tomcat-6.0.33/bin/startup.sh
```

After a successful startup, the default web applications included with Tomcat will be available by visiting **http://localhost:8080/**. If everything is fine then it should display following result:



Further information about configuring and running Tomcat can be found in the documentation included here, as well as on the Tomcat web site: <http://tomcat.apache.org>

Tomcat can be stopped by executing the following commands on windows machine:

```
%CATALINA_HOME%\bin\shutdown
or
C:\apache-tomcat-5.5.29\bin\shutdown
```

Tomcat can be stopped by executing the following commands on Unix (Solaris, Linux, etc.) machine:

```
$CATALINA_HOME/bin/shutdown.sh
or
/usr/local/apache-tomcat-5.5.29/bin/shutdown.sh
```

Step 3 - Setup Eclipse (IDE)

All the examples in this tutorial has been written using Eclipse IDE. So I would suggest you have latest version of Eclipse installed on your machine.

To install Eclipse dDownload the latest Eclipse binaries from <http://www.eclipse.org/downloads/>. Once you downloaded the installation, unpack the binary distribution into a convenient location. For example in C:\eclipse on windows, or /usr/local/eclipse on Linux/Unix and finally set PATH variable appropriately.

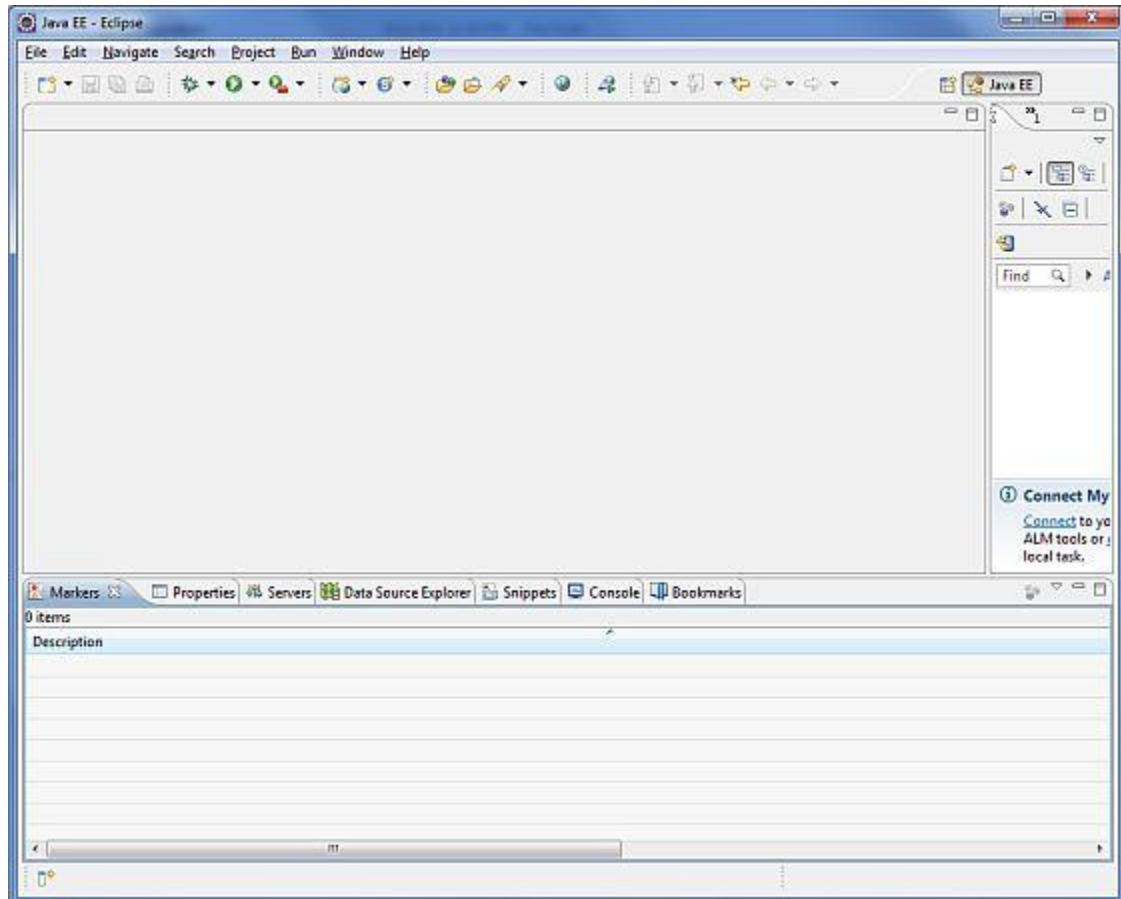
Eclipse can be started by executing the following commands on windows machine, or you can simply double click on eclipse.exe

```
%C:\eclipse\eclipse.exe
```

Eclipse can be started by executing the following commands on Unix (Solaris, Linux, etc.) machine:

```
$/usr/local/eclipse/eclipse
```

After a successful startup, if everything is fine then it should display following result:



Step 4 - Setup Struts2 Libraries

Now if everything is fine, then you can proceed to setup your Struts 2 framework. Following are the simple steps to download and install Struts2 on your machine.

- Make a choice whether you want to install Hibernate on Windows, or Unix and then proceed to the next step to download .zip file for windows and .tar file for Unix.
- Download the latest version of Struts2 binaries from <http://struts.apache.org/download.cgi>.
- At the time of writing this tutorial, I downloaded **struts-2.0.14-all.zip** and when you unzip the downloaded file it will give you directory structure inside C:\struts-2.2.3 as follows.

Name	Date modified	Type	Size
apps	4/8/2011 9:30 AM	File folder	
docs	4/8/2011 9:27 AM	File folder	
lib	4/8/2011 9:30 AM	File folder	
src	4/8/2011 9:30 AM	File folder	
ANTLR-LICENSE	4/8/2011 8:53 AM	Text Document	2 KB
CLASSWORLDS-LICENSE	4/8/2011 8:53 AM	Text Document	2 KB
FREEMARKER-LICENSE	4/8/2011 8:53 AM	Text Document	3 KB
LICENSE	4/8/2011 8:52 AM	Text Document	10 KB
NOTICE	4/8/2011 8:52 AM	Text Document	1 KB
OGNL-LICENSE	4/8/2011 8:53 AM	Text Document	3 KB
OVAL-LICENSE	4/8/2011 8:53 AM	Text Document	12 KB
SITEMESH-LICENSE	4/8/2011 8:53 AM	Text Document	3 KB
XPP3-LICENSE	4/8/2011 8:53 AM	Text Document	3 KB
XSTREAM-LICENSE	4/8/2011 8:53 AM	Text Document	2 KB

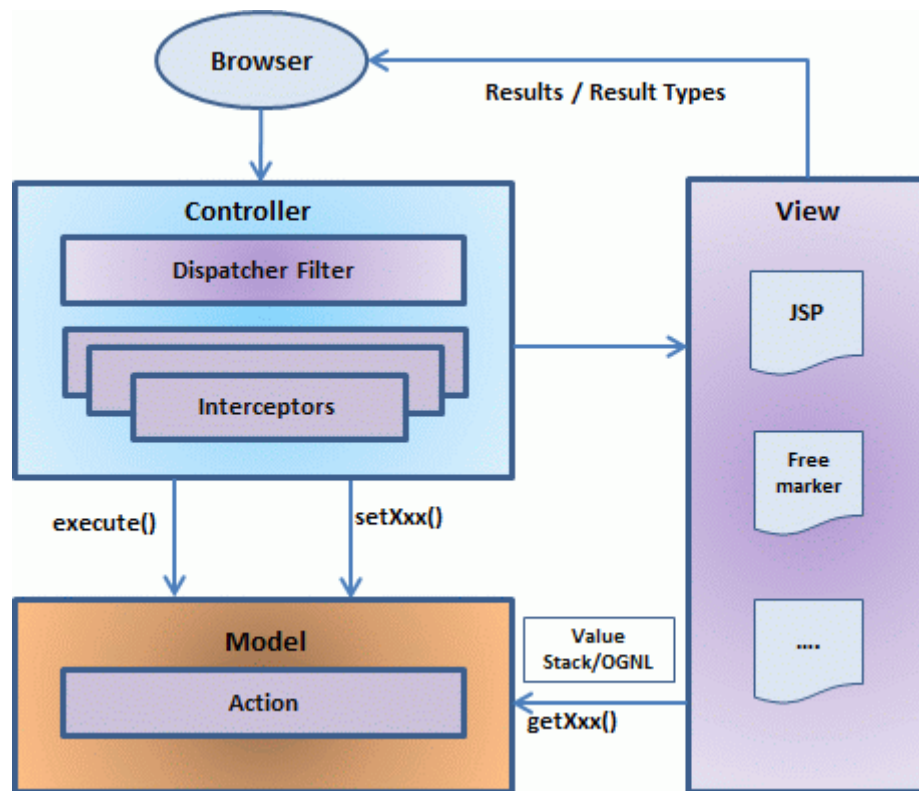
Second step is to extract the zip file in any location, I downloaded & extracted **struts-2.2.3-all.zip** in **c:** folder on my Windows 7 machine so that I have all the jar files into **C:\struts-2.2.3\lib**. Make sure you set your CLASSPATH variable properly otherwise you will face problem while running your application.

Struts 2 Architecture:

From a high level, Struts2 is a pull-MVC (or MVC2) framework. The Model-View-Controller pattern in Struts2 is realized with following five core components:

1. **Actions**
2. **Interceptors**
3. **Value Stack / OGNL**
4. **Results / Result types**
5. **View technologies**

Struts 2 is slightly different from a traditional MVC framework in that the action takes the role of the model rather than the controller, although there is some overlap.



The above diagram depicts the **M**odel, **V**iew and **C**ontroller to the Struts2 high level architecture. The controller is implemented with a Struts2 dispatch servlet filter as well as interceptors, the model is implemented with actions, and the view as a combination of result types and results. The value stack and OGNL provide common thread, linking and enabling integration between the other components.

Apart from the above components, there will be a lot of information that relates to configuration. Configuration for the web application, as well as configuration for actions, interceptors, results, etc.

This is the architectural overview of the Struts 2 MVC pattern. We will go through each component in more detail in the subsequent chapters.

Request life cycle:

Based on the above diagram, one can explain the user's request life cycle in Struts 2 as follows:

1. User sends a request to the server for requesting for some resource (i.e pages).
2. The FilterDispatcher looks at the request and then determines the appropriate Action.
3. Configured interceptors functionalities applies such as validation, file upload etc.
4. Selected action is executed to perform the requested operation.
5. Again, configured interceptors are applied to do any post-processing if required.
6. Finally the result is prepared by the view and returns the result to the user.

Struts 2 Hello World Example:

As you learnt from the Struts 2 architecture, when you click on a hyperlink or submit an HTML form in a Struts 2 web application, the input is collected by the Controller which is sent to a Java class called Actions. After the Action is executed, a Result selects a resource to render the response. The resource is generally a JSP, but it can also be a PDF file, an Excel spreadsheet, or a Java applet window.

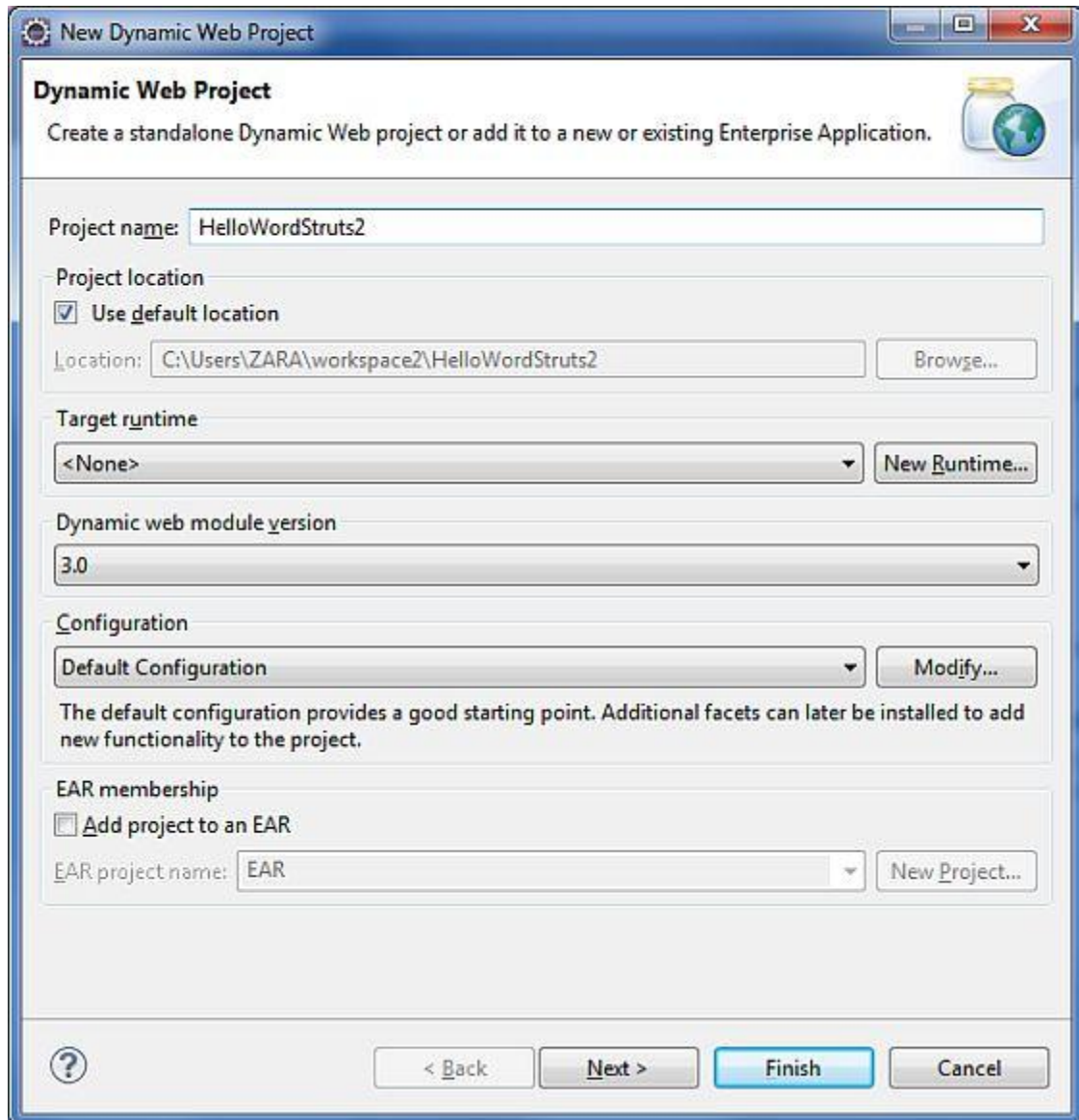
Assume you already build-up your development environment. Now let us proceed for building our first **Hello World** struts2 project. The aim of this project is to build a web application that collects the user's name and displays "Hello World" followed by the user name. We would have to create following four components for any Struts 2 project:

SN	Components & Description
1	Action Create an action class which will contain complete business logic and control the interaction between the user, the model, and the view.
2	Interceptors Create interceptors if required, or use existing interceptors. This is part of Controller.
3	View Create a JSPs to interact with the user to take input and to present the final messages.
4	Configuration Files Create configuration files to couple the Action, View and Controllers. These files are struts.xml, web.xml, struts.properties.

I am going to use Eclipse IDE, so all the required components will be created under a Dynamic Web Project. So let us start with creating Dynamic Web Project.

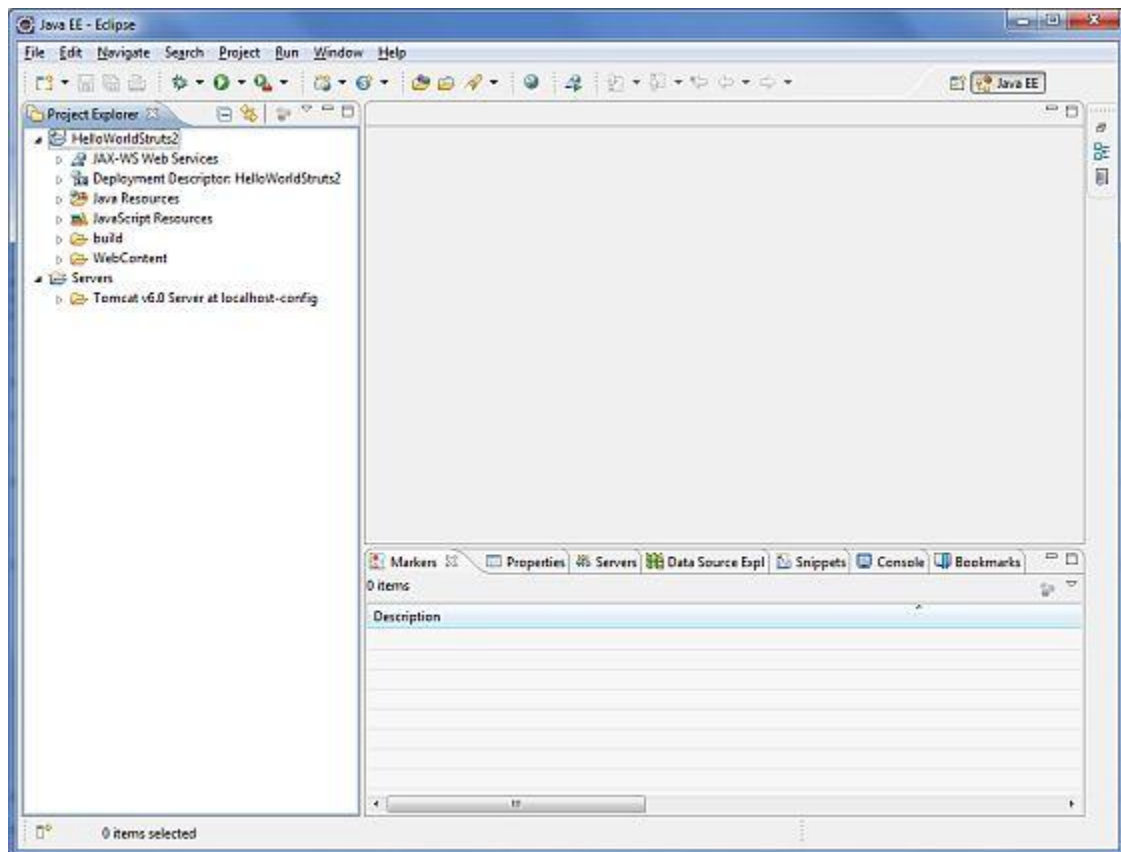
Create a Dynamic Web Project:

Start your Eclipse and then go with **File > New > Dynamic Web Project** and enter project name as **HelloWorldStruts2** and set rest of the options as given in the following screen:



The screenshot shows the 'New Dynamic Web Project' dialog box in Eclipse. The title bar reads 'New Dynamic Web Project'. The main heading is 'Dynamic Web Project' with a subtitle 'Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.' and a small globe icon. The 'Project name' field contains 'HelloWordStruts2'. The 'Project location' section has a checked 'Use default location' checkbox and a 'Location' field with the path 'C:\Users\ZARA\workspace2\HelloWordStruts2' and a 'Browse...' button. The 'Target runtime' section has a dropdown menu set to '<None>' and a 'New Runtime...' button. The 'Dynamic web module version' dropdown is set to '3.0'. The 'Configuration' section has a dropdown set to 'Default Configuration' and a 'Modify...' button, with a note below stating: 'The default configuration provides a good starting point. Additional facets can later be installed to add new functionality to the project.' The 'EAR membership' section has an unchecked 'Add project to an EAR' checkbox, an 'EAR project name' field with 'EAR', and a 'New Project...' button. At the bottom, there is a help icon (?), and four buttons: '< Back', 'Next >', 'Finish' (highlighted in blue), and 'Cancel'.

Select all the default options in the next screens and finally check **Generate Web.xml deployment descriptor** option. This will create a dynamic web project for you in Eclipse. Now go with **Windows > Show View > Project Explorer**, and you will see your project window something as below:



Now copy following files from struts 2 lib folder **C:\struts-2.2.3\lib** to our project's **WEB-INF\lib** folder. To do this, you can simply drag and drop all the following files into WEB-INF\lib folder.

- commons-fileupload-x.y.z.jar
- commons-io-x.y.z.jar
- commons-lang-x.y.jar
- commons-logging-x.y.z.jar
- commons-logging-api-x.y.jar
- freemarker-x.y.z.jar
- javassist-.xy.z.GA
- ognl-x.y.z.jar
- struts2-core-x.y.z.jar
- xwork-core.x.y.z.jar

Create Action Class:

Action class is the key to Struts 2 application and we implement most of the business logic in action class. So let us create a java file HelloWorldAction.java under **Java Resources > src** with a package name **com.tutorialspoint.struts2** with the contents given below.

The Action class responds to a user action when user clicks a URL. One or more of the Action class's methods are executed and a String result is returned. Based on the value of the result, a specific JSP page is rendered.

```
package com.tutorialspoint.struts2;

public class HelloWorldAction{
```

```
private String name;

public String execute() throws Exception {
    return "success";
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
}
```

This is a very simple class with one property called "name". We have standard getters and setter methods for the "name" property and an execute method that returns the string "success".

The Struts 2 framework will create an object of the HelloWorldAction class and call the execute method in response to a user's action. You put your business logic inside execute method and finally returns the String constant. Simply saying for each URL, you would have to implement one action class and either you can use that class name directly as your action name or you can map to some other name using struts.xml file as shown below.

Create a View

We need a JSP to present the final message, this page will be called by Struts 2 framework when a predefined action will happen and this mapping will be defined in struts.xml file. So let us create the below jsp file **HelloWorld.jsp** in the WebContent folder in your eclipse project. To do this, right click on the WebContent folder in the project explorer and select **New > JSP File**.

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Hello World</title>
</head>
<body>
    Hello World, <s:property value="name"/>
</body>
</html>
```

The taglib directive tells the Servlet container that this page will be using the Struts 2 tags and that these tags will be preceded by s. The s:property tag displays the value of action class property "name" which is returned by the method **getName()** of the HelloWorldAction class.

Create main page:

We also need to create **index.jsp** in the WebContent folder. This file will serve as the initial action URL where a user can click to tell the Struts 2 framework to call the a defined method of the HelloWorldAction class and render the HelloWorld.jsp view.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
```

```
<head>
<title>Hello World</title>
</head>
<body>
  <h1>Hello World From Struts2</h1>
  <form action="hello">
    <label for="name">Please enter your name</label><br/>
    <input type="text" name="name"/>
    <input type="submit" value="Say Hello"/>
  </form>
</body>
</html>
```

The **hello** action defined in the above view file will be mapped to the HelloWorldAction class and its execute method using struts.xml file. When a user clicks on the Submit button it will cause the Struts 2 framework to run the execute method defined in the HelloWorldAction class and based on the returned value of the method, an appropriate view will be selected and rendered as a response.

Configuration Files

We need a mapping to tie the URL, the HelloWorldAction class (Model), and the HelloWorld.jsp (the view) together. The mapping tells the Struts 2 framework which class will respond to the user's action (the URL), which method of that class will be executed, and what view to render based on the String result that method returns.

So let us create a file called **struts.xml**. Since Struts 2 requires struts.xml to be present in classes folder. So create struts.xml file under the WebContent/WEB-INF/classes folder. Eclipse does not create the "classes" folder by default, so you need to do this yourself. To do this, right click on the WEB-INF folder in the project explorer and select **New > Folder**. Your struts.xml should look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
  "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
  <constant name="struts.devMode" value="true" />
  <package name="helloworld" extends="struts-default">

    <action name="hello"
      class="com.tutorialspoint.struts2.HelloWorldAction"
      method="execute">
      <result name="success">/HelloWorld.jsp</result>
    </action>
  </package>
</struts>
```

Few words about the above configuration file. Here we set the constant **struts.devMode** to **true**, because we are working in development environment and we need to see some useful log messages. Then, we defined a package called **helloworld**. Creating a package is useful when you want to group your actions together. In our example, we named our action as "hello" which is corresponding to the URL **/hello.action** and is backed up by the **HelloWorldAction.class**. The **execute** method of **HelloWorldAction.class** is the method that is run when the URL/**hello.action** is invoked. If the outcome of the **execute** method returns "success", then we take the user to **HelloWorld.jsp**.

Next step is to create a **web.xml** file which is an entry point for any request to Struts 2. The entry point of Struts2 application will be a filter defined in deployment descriptor (web.xml). Hence we will define an entry of org.apache.struts2.dispatcher.FilterDispatcher class in web.xml. The web.xml file needs to be created under the WEB-INF folder under WebContent. Eclipse had

already created a skeleton web.xml file for you when you created the project. So, let's just modify it as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  id="WebApp_ID" version="3.0">

  <display-name>Struts 2</display-name>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <filter>
    <filter-name>struts2</filter-name>
    <filter-class>
      org.apache.struts2.dispatcher.FilterDispatcher
    </filter-class>
  </filter>

  <filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-app>
```

We have specified index.jsp to be our welcome file. Then we have configured the Struts2 filter to run on all urls (i.e, any url that match the pattern /*)

Enable Detailed Log:

You can enable complete logging functionality while working with Struts 2 by creating **logging.properties** file under **WEB-INF/classes** folder. Keep the following two lines in your property file:

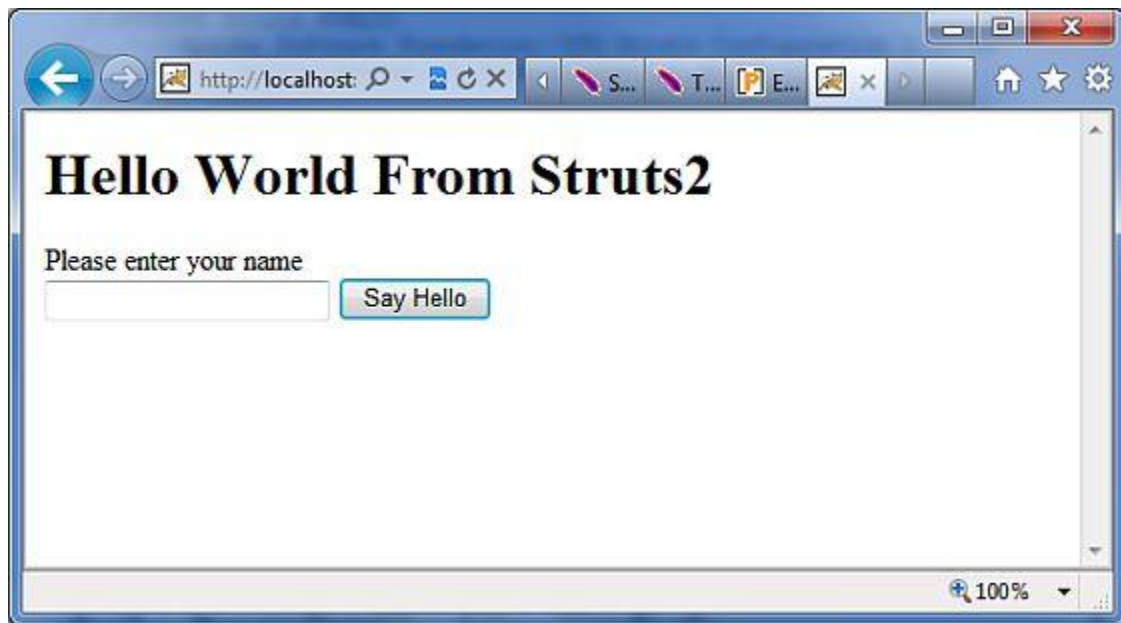
```
org.apache.catalina.core.ContainerBase.[Catalina].level = INFO
org.apache.catalina.core.ContainerBase.[Catalina].handlers = \
    java.util.logging.ConsoleHandler
```

The default logging.properties specifies a ConsoleHandler for routing logging to stdout and also a FileHandler. A handler's log level threshold can be set using SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST or ALL.

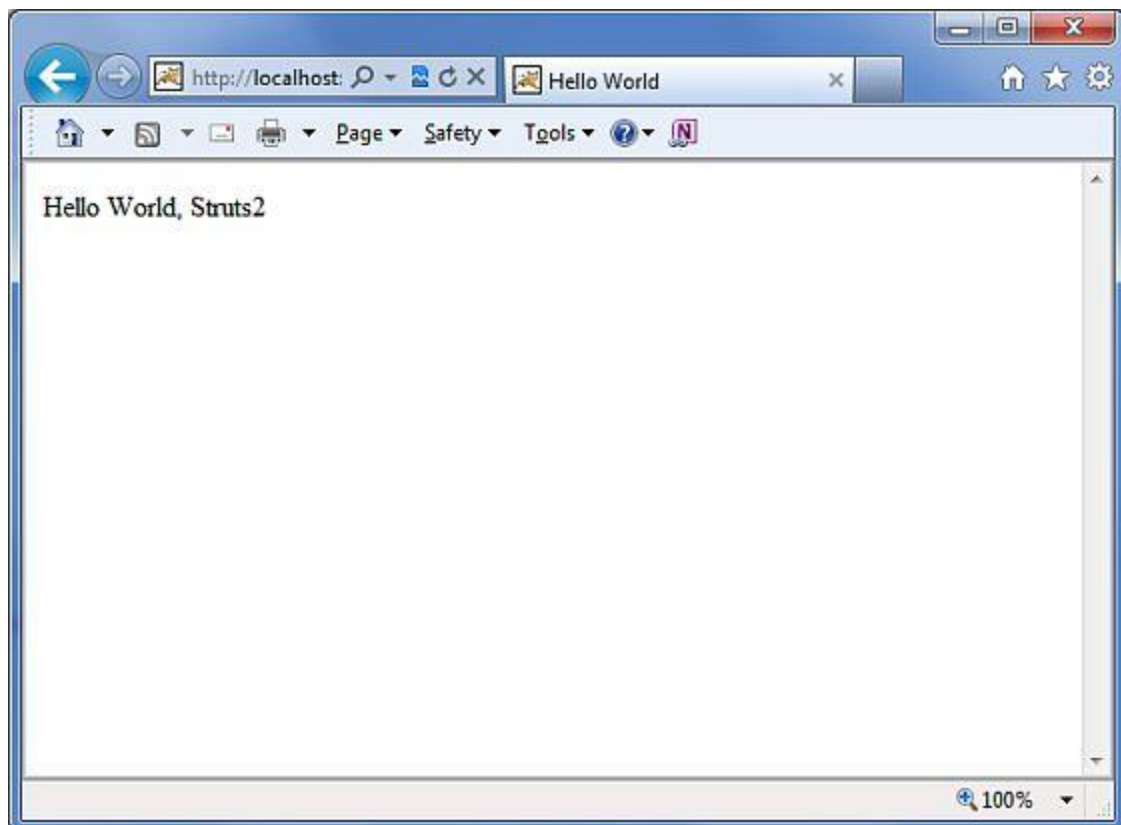
That's it. We are ready to run our Hello World application using Struts 2 framework.

Execute the Application

Right click on the project name and click **Export > WAR File** to create a War file. Then deploy this WAR in the Tomcat's webapps directory. Finally, start Tomcat server and try to access URL <http://localhost:8080/HelloWorldStruts2/index.jsp>. This will give you following screen:



Enter a value "Struts2" and submit the page. You should see the next page



Note that you can define **index** as an action in struts.xml file and in that case you can call index page as `http://localhost:8080/HelloWorldStruts2/index.action`. Check below how you can define index as an action:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
```

```
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
<constant name="struts.devMode" value="true" />
  <package name="helloworld" extends="struts-default">

    <action name="index">
      <result >/index.jsp</result>
    </action>

    <action name="hello"
      class="com.tutorialspoint.struts2.HelloWorldAction"
      method="execute">
      <result name="success">/HelloWorld.jsp</result>
    </action>

  </package>
</struts>
```

Struts 2 Configuration Files

This chapter will take you through basic configuration required for a Struts 2 application. Here we will see what will be configured in few important configuration files : **web.xml**, **struts.xml**, **struts-config.xml** and **struts.properties**

Honestly speaking you can survive using web.xml and struts.xml configuration files and you have seen in previous chapter that our example worked using these two files, but for your knowledge let me explain other files as well.

The web.xml file:

The web.xml configuration file is a J2EE configuration file that determines how elements of the HTTP request are processed by the servlet container. It is not strictly a Struts2 configuration file, but it is a file that needs to be configured for Struts2 to work.

As discussed earlier, this file provides an entry point for any web application. The entry point of Struts2 application will be a filter defined in deployment descriptor (web.xml). Hence we will define an entry of *FilterDispatcher* class in web.xml. The web.xml file needs to be created under the folder **WebContent/WEB-INF**.

This is the first configuration file you will need to configure if you are starting without the aid of a template or tool that generates it (such as Eclipse or Maven2). Following is the content of web.xml file which we used in our last example.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  id="WebApp_ID" version="3.0">

  <display-name>Struts 2</display-name>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <filter>
    <filter-name>struts2</filter-name>
    <filter-class>
```



```
        org.apache.struts2.dispatcher.FilterDispatcher
    </filter-class>
</filter>

<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

</web-app>
```

Note that we map the Struts 2 filter to **/***, and not to **/*.action** which means that all urls will be parsed by the struts filter. We will cover this when we will go through the Annotations chapter.

The struts.xml file:

The **struts.xml** file contains the configuration information that you will be modifying as actions are developed. This file can be used to override default settings for an application, for example *struts.devMode = false* and other settings which are defined in property file. This file can be created under the folder **WEB-INF/classes**.

Let us have a look at the struts.xml file we created in the Hello World example explained in previous chapter.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <constant name="struts.devMode" value="true" />
    <package name="helloworld" extends="struts-default">

        <action name="hello"
            class="com.tutorialspoint.struts2.HelloWorldAction"
            method="execute">
            <result name="success">/HelloWorld.jsp</result>
        </action>
        <!-- more actions can be listed here -->

    </package>
    <!-- more packages can be listed here -->
</struts>
```

The first thing to note is the **DOCTYPE**. All struts configuration file need to have the correct doctype as shown in our little example. **<struts>** is the root tag element, under which we declare different packages using **<package>** tags. Here **<package>** allows separation and modularization of the configuration. This is very useful when you have a large project and project is divided into different modules.

Say, if your project has three domains - *business_applicaition*, *customer_application* and *staff_application*, you could create three packages and store associated actions in the appropriate package. The package tag has the following attributes:

Attribute	Description
-----------	-------------

name (required)	The unique identifier for the package
extends	Which package does this package extend from? By default, we use struts-default as the base package.
abstract	If marked true, the package is not available for end user consumption.
namespace	Unique namespace for the actions

The **constant** tag along with name and value attributes will be used to override any of the following properties defined in **default.properties**, like we just set **struts.devMode** property. Setting **struts.devMode** property allows us to see more debug messages in the log file.

We define **action** tags corresponds to every URL we want to access and we define a class with `execute()` method which will be accessed whenever we will access corresponding URL.

Results determine what gets returned to the browser after an action is executed. The string returned from the action should be the name of a result. Results are configured per-action as above, or as a "global" result, available to every action in a package. Results have optional **name** and **type** attributes. The default name value is "success".

Struts.xml file can grow big over time and so breaking it by packages is one way of modularizing it, but struts offers another way to modularize the struts.xml file. You could split the file into multiple xml files and import them in the following fashion.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <include file="my-struts1.xml"/>
    <include file="my-struts2.xml"/>
</struts>
```

The other configuration file that we haven't covered is the struts-default.xml. This file contains the standard configuration settings for Struts and you would not have to touch these settings for 99.99% of your projects. For this reason, we are not going into too much detail on this file. If you are interested, take a look into the at the **default.properties** file available in struts2-core-2.2.3.jar file.

The struts-config.xml file:

The struts-config.xml configuration file is a link between the View and Model components in the Web Client but you would not have to touch these settings for 99.99% of your projects. The configuration file basically contains following main elements:

SN	Interceptor & Description
1	struts-config This is the root node of the configuration file.
2	form-beans

	This is where you map your ActionForm subclass to a name. You use this name as an alias for your ActionForm throughout the rest of the struts-config.xml file, and even on your JSP pages.
3	global forwards This section maps a page on your webapp to a name. You can use this name to refer to the actual page. This avoids hardcoding URLs on your web pages.
4	action-mappings This is where you declare form handlers and they are also known as action mappings .
5	controller This section configures Struts internals and rarely used in practical situations.
6	plug-in This section tells Struts where to find your properties files, which contain prompts and error messages

Following is the sample struts-config.xml file:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 1.0//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_0.dtd">

<struts-config>

    <!-- ===== Form Bean Definitions ===== -->
    <form-beans>
        <form-bean name="login" type="test.struts.LoginForm" />
    </form-beans>

    <!-- ===== Global Forward Definitions ===== -->
    <global-forwards>
    </global-forwards>

    <!-- ===== Action Mapping Definitions ===== -->
    <action-mappings>
        <action
            path="/login"
            type="test.struts.LoginAction" >

            <forward name="valid" path="/jsp/MainMenu.jsp" />
            <forward name="invalid" path="/jsp/LoginView.jsp" />
        </action>
    </action-mappings>

    <!-- ===== Controller Definitions ===== -->
    <controller
        contentType="text/html; charset=UTF-8"
        debug="3"
        maxFileSize="1.618M"
        locale="true"
        nocache="true"/>
    </controller>
</struts-config>
```

For more detail on struts-config.xml file, kindly check your struts documentation.

The struts.properties file

This configuration file provides a mechanism to change the default behavior of the framework. Actually all of the properties contained within the **struts.properties** configuration file can also be configured in the **web.xml** using the **init-param**, as well using the **constant** tag in the **struts.xml** configuration file. But if you like to keep the things separate and more struts specific then you can create this file under the folder **WEB-INF/classes**.

The values configured in this file will override the default values configured in **default.properties** which is contained in the struts2-core-x.y.z.jar distribution. There are a couple of properties that you might consider changing using **struts.properties** file:

```
### When set to true, Struts will act much more friendly for developers
struts.devMode = true

### Enables reloading of internationalization files
struts.i18n.reload = true

### Enables reloading of XML configuration files
struts.configuration.xml.reload = true

### Sets the port that the server is run on
struts.url.http.port = 8080
```

Here any line starting with hash (#) will be assumed as a comment and it will be ignored by Struts 2.

Struts 2 Actions:

Actions are the core of the Struts2 framework, as they are for any MVC (Model View Controller) framework. Each URL is mapped to a specific action, which provides the processing logic necessary to service the request from the user.

But the action also serves in two other important capacities. First, the action plays an important role in the transfer of data from the request through to the view, whether its a JSP or other type of result. Second, the action must assist the framework in determining which result should render the view that will be returned in the response to the request.

Create Action:

The only requirement for actions in Struts2 is that there must be one no-argument method that returns either a String or Result object and must be a POJO. If the no-argument method is not specified, the default behavior is to use the `execute()` method.

Optionally you can extend the **ActionSupport** class which implements six interfaces including **Action** interface. The Action interface is as follows:

```
public interface Action {
    public static final String SUCCESS = "success";
    public static final String NONE = "none";
    public static final String ERROR = "error";
    public static final String INPUT = "input";
    public static final String LOGIN = "login";
    public String execute() throws Exception;
}
```

Let us take a look at the action method in the Hello World example:

```
package com.tutorialspoint.struts2;

public class HelloWorldAction{
    private String name;

    public String execute() throws Exception {
        return "success";
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

To illustrate the point that the action method controls the view, let us make the following change to the **execute** method and extend the class `ActionSupport` as follows:

```
package com.tutorialspoint.struts2;

import com.opensymphony.xwork2.ActionSupport;

public class HelloWorldAction extends ActionSupport{
    private String name;

    public String execute() throws Exception {
        if ("SECRET".equals(name))
        {
            return SUCCESS;
        }else{
            return ERROR;
        }
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

In this example, we have some logic in the execute method to look at the name attribute. If the attribute equals to the string "SECRET", we return SUCCESS as the result otherwise we return ERROR as the result. Because we have extended `ActionSupport`, so we can use String constants SUCCESS and ERROR. Now, let us modify our struts.xml file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <constant name="struts.devMode" value="true" />
    <package name="helloworld" extends="struts-default">
        <action name="hello">
```

```
class="com.tutorialspoint.struts2.HelloWorldAction"  
method="execute">  
    <result name="success">/HelloWorld.jsp</result>  
    <result name="error">/AccessDenied.jsp</result>  
</action>  
</package>  
</struts>
```

Create a View

Let us create the below jsp file **HelloWorld.jsp** in the WebContent folder in your eclipse project. To do this, right click on the WebContent folder in the project explorer and select **New >JSP File**. This file will be called in case return result is SUCCESS which is a String constant "success" as defined in Action interface:

```
<%@ page contentType="text/html; charset=UTF-8" %>  
<%@ taglib prefix="s" uri="/struts-tags" %>  
<html>  
<head>  
<title>Hello World</title>  
</head>  
<body>  
    Hello World, <s:property value="name"/>  
</body>  
</html>
```

Following is the file which will be invoked by the framework in case action result is ERROR which is equal to String constant "error". Following is the content of **AccessDenied.jsp**

```
<%@ page contentType="text/html; charset=UTF-8" %>  
<%@ taglib prefix="s" uri="/struts-tags" %>  
<html>  
<head>  
<title>Access Denied</title>  
</head>  
<body>  
    You are not authorized to view this page.  
</body>  
</html>
```

We also need to create **index.jsp** in the WebContent folder. This file will serve as the initial action URL where the user can click to tell the Struts 2 framework to call the **execute** method of the HelloWorldAction class and render the HelloWorld.jsp view.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"  
    pageEncoding="ISO-8859-1"%>  
<%@ taglib prefix="s" uri="/struts-tags"%>  
    <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  
    "http://www.w3.org/TR/html4/loose.dtd">  
<html>  
<head>  
<title>Hello World</title>  
</head>  
<body>  
    <h1>Hello World From Struts2</h1>  
    <form action="hello">  
        <label for="name">Please enter your name</label><br/>  
        <input type="text" name="name"/>  
        <input type="submit" value="Say Hello"/>  
    </form>
```

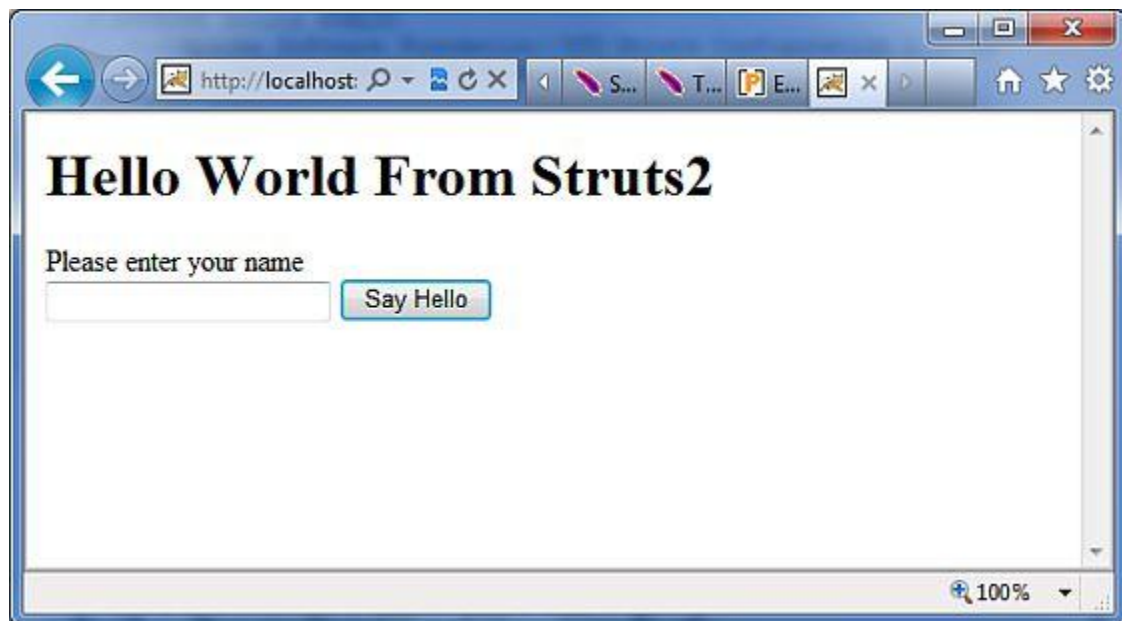


```
</body>  
</html>
```

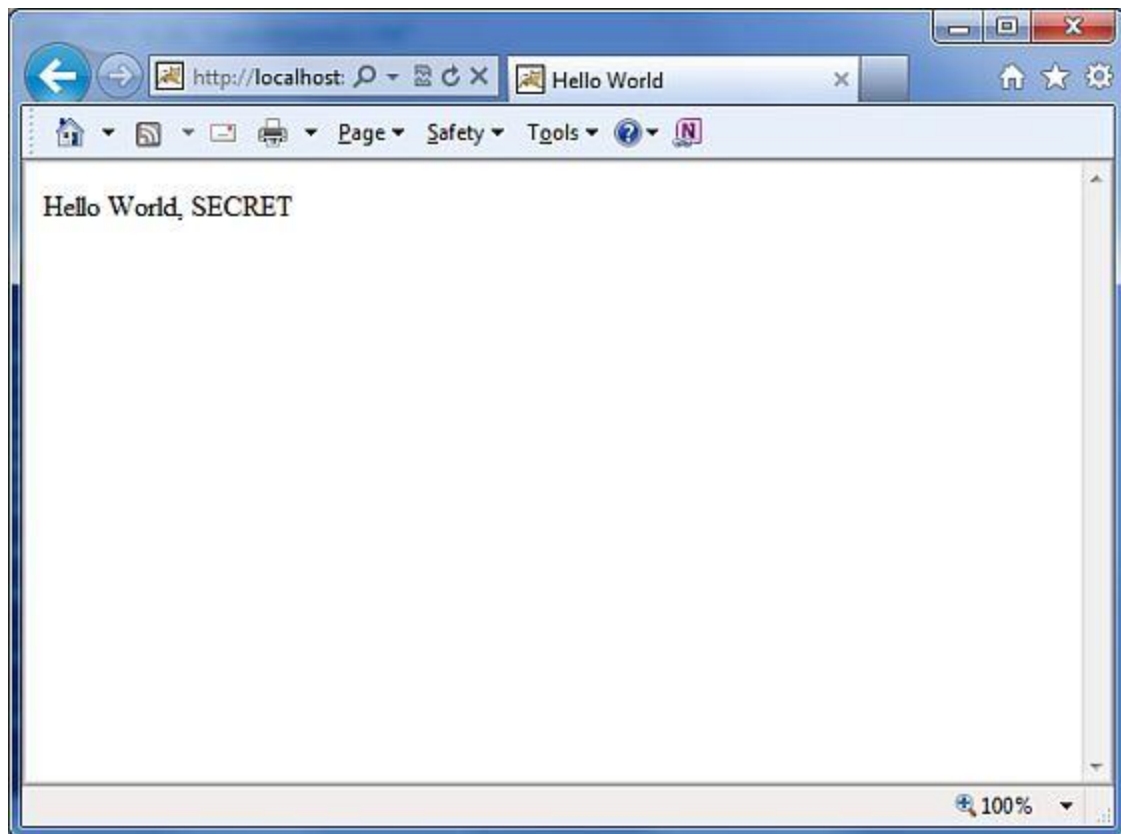
That's it, there is no change required web.xml file, so let us use same web.xml which we had created in Examples chapter. Now we are ready to run our Hello World application using Struts 2 framework.

Execute the Application

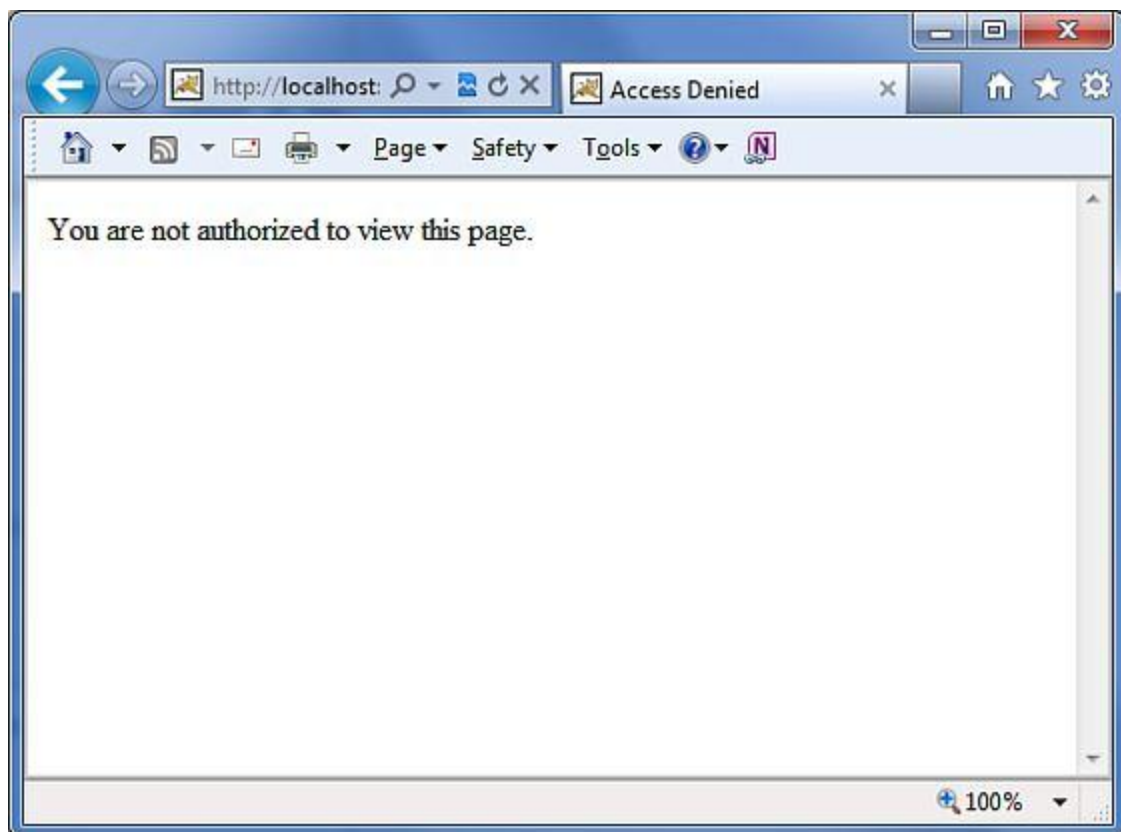
Right click on the project name and click **Export > WAR File** to create a War file. Then deploy this WAR in the Tomcat's webapps directory. Finally, start Tomcat server and try to access URL <http://localhost:8080/HelloWorldStruts2/index.jsp>. This will give you following screen:



Let us enter a word as "SECRET" and you should see the following page:



Now enter any word other than "SECRET" and you should see the following page:



Create Multiple Actions:

You will frequently define more than one actions to handle different requests and to provide different URLs to the users, accordingly you will define different classes as defined below:

```
package com.tutorialspoint.struts2;
import com.opensymphony.xwork2.ActionSupport;

class MyAction extends ActionSupport{
    public static String GOOD = SUCCESS;
    public static String BAD = ERROR;
}

public class HelloWorld extends ActionSupport{
    ...
    public String execute()
    {
        if ("SECRET".equals(name)) return MyAction.GOOD;
        return MyAction.BAD;
    }
    ...
}

public class SomeOtherClass extends ActionSupport{
    ...
    public String execute()
    {
        return MyAction.GOOD;
    }
    ...
}
```

You will configure these actions in struts.xml file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <constant name="struts.devMode" value="true" />
    <package name="helloworld" extends="struts-default">
        <action name="hello"
            class="com.tutorialspoint.struts2.HelloWorld"
            method="execute">
            <result name="success">/HelloWorld.jsp</result>
            <result name="error">/AccessDenied.jsp</result>
        </action>
        <action name="something"
            class="com.tutorialspoint.struts2.SomeOtherClass"
            method="execute">
            <result name="success">/Something.jsp</result>
            <result name="error">/AccessDenied.jsp</result>
        </action>
    </package>
</struts>
```

As you can see in the above hypothetical example, the action results SUCCESS and ERROR are duplicated. To get around this issue, it is suggested that you create a class that contain the result outcomes.

Struts 2 Interceptors:

Interceptors are conceptually the same as servlet filters or the JDKs Proxy class. Interceptors allow for crosscutting functionality to be implemented separately from the action as well as the framework. You can achieve the following using interceptors:

- Providing preprocessing logic before the action is called.
- Providing postprocessing logic after the action is called.
- Catching exceptions so that alternate processing can be performed.

Many of the features provided in the Struts2 framework are implemented using interceptors; examples include exception handling, file uploading, lifecycle callbacks and validation etc. In fact, as Struts2 bases much of its functionality on interceptors, it is not unlikely to have 7 or 8 interceptors assigned per action.

Struts2 Framework Interceptors:

Struts 2 framework provides a good list of out-of-the-box interceptors that come preconfigured and ready to use. Few of the important interceptors are listed below:

SN	Interceptor & Description
1	alias Allows parameters to have different name aliases across requests.
2	checkbox Assists in managing check boxes by adding a parameter value of false for check boxes that are not checked.
3	conversionError Places error information from converting strings to parameter types into the action's field errors.
4	createSession Automatically creates an HTTP session if one does not already exist.
5	debugging Provides several different debugging screens to the developer.
6	execAndWait Sends the user to an intermediary waiting page while the action executes in the background.
7	exception Maps exceptions that are thrown from an action to a result, allowing automatic exception handling via redirection.
8	fileUpload Facilitates easy file uploading.
9	i18n Keeps track of the selected locale during a user's session.

10	logger Provides simple logging by outputting the name of the action being executed.
11	params Sets the request parameters on the action.
12	prepare This is typically used to do pre-processing work, such as setup database connections.
13	profile Allows simple profiling information to be logged for actions.
14	scope Stores and retrieves the action's state in the session or application scope.
15	ServletConfig Provides the action with access to various servlet-based information.
16	timer Provides simple profiling information in the form of how long the action takes to execute.
17	token Checks the action for a valid token to prevent duplicate formsubmission.
18	validation Provides validation support for actions

Please look into Struts 2 documentation for complete detail on above mentioned interceptors. But I will show you how to use an interceptor in general in your Struts application.

How to use Interceptors?

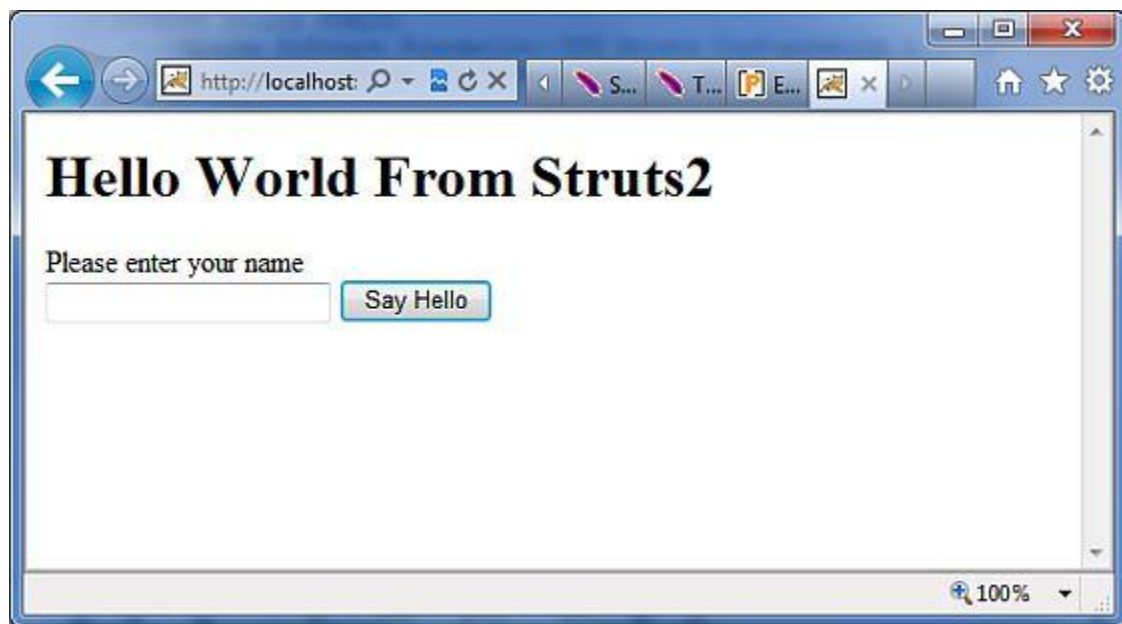
Let us see how to use an already existing interceptor to our "Hello World" program. We will use the **timer** interceptor whose purpose is to measure how long it took to execute an action method. Same time I'm using **params** interceptor whose purpose is to send the request parameters to the action. You can try your example without using this interceptor and you will find that **name** property is not being set because parameter is not able to reach to the action.

We will keep HelloWorldAction.java, web.xml, HelloWorld.jsp and index.jsp files as they have been created in Examples chapter but let us modify the **struts.xml** file to add an interceptor as follows

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <constant name="struts.devMode" value="true" />
    <package name="helloworld" extends="struts-default">
        <action name="hello"
```

```
class="com.tutorialspoint.struts2.HelloWorldAction"  
method="execute">  
  <interceptor-ref name="params"/>  
  <interceptor-ref name="timer" />  
  <result name="success">/HelloWorld.jsp</result>  
</action>  
</package>  
</struts>
```

Right click on the project name and click **Export > WAR File** to create a War file. Then deploy this WAR in the Tomcat's webapps directory. Finally, start Tomcat server and try to access URL <http://localhost:8080/HelloWorldStruts2/index.jsp>. This will give you following screen:



Now enter any word in the given text box and click Say Hello button to execute the defined action. Now if you will check the log generated, you will find following text:

```
INFO: Server startup in 3539 ms  
27/08/2011 8:40:53 PM  
com.opensymphony.xwork2.util.logging.commons.CommonsLogger info  
INFO: Executed action [/hello!execute] took 109 ms.
```

Here bottom line is being generated because of **timer** interceptor which is telling that action took total 109ms to be executed.

Create Custom Interceptors

Using custom interceptors in your application is an elegant way to provide cross-cutting application features. Creating a custom interceptor is easy; the interface that needs to be extended is the following **Interceptor** interface:

```
public interface Interceptor extends Serializable{  
    void destroy();  
    void init();  
    String intercept(ActionInvocation invocation)
```



```
throws Exception;  
}
```

As the names suggest, the `init()` method provides a way to initialize the interceptor, and the `destroy()` method provides a facility for interceptor cleanup. Unlike actions, interceptors are reused across requests and need to be thread-safe, especially the `intercept()` method.

The **ActionInvocation** object provides access to the runtime environment. It allows access to the action itself and methods to invoke the action and determine whether the action has already been invoked.

If you have no need for initialization or cleanup code, the **AbstractInterceptor** class can be extended. This provides a default no-operation implementation of the `init()` and `destroy()` methods.

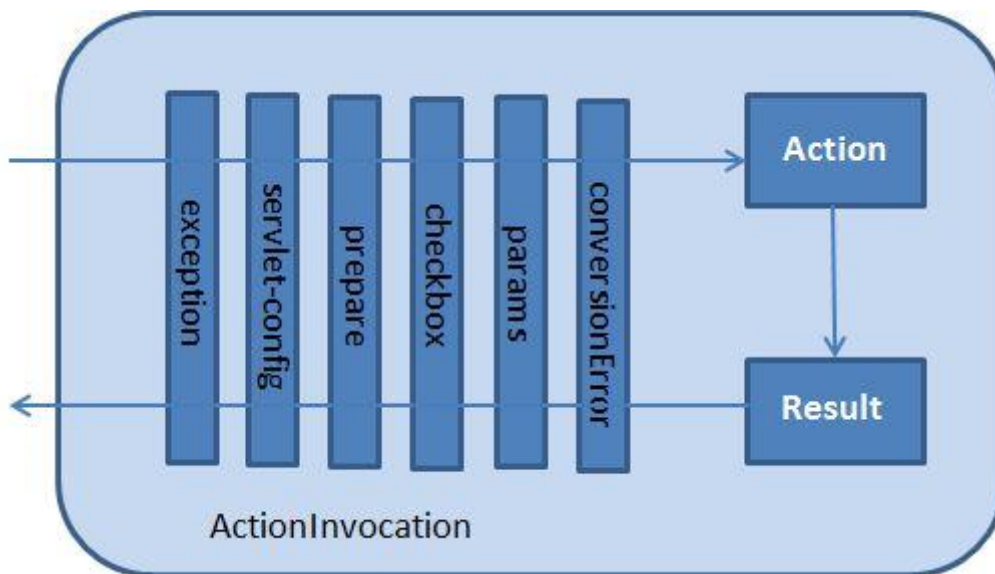
Create Interceptor Class:

Let us create following **MyInterceptor.java** in **Java Resources > src** folder:

```
package com.tutorialspoint.struts2;  
  
import java.util.*;  
import com.opensymphony.xwork2.ActionInvocation;  
import com.opensymphony.xwork2.interceptor.AbstractInterceptor;  
  
public class MyInterceptor extends AbstractInterceptor {  
  
    public String intercept(ActionInvocation invocation) throws Exception {  
  
        /* let us do some pre-processing */  
        String output = "Pre-Processing";  
        System.out.println(output);  
  
        /* let us call action or next interceptor */  
        String result = invocation.invoke();  
  
        /* let us do some post-processing */  
        output = "Post-Processing";  
        System.out.println(output);  
  
        return result;  
  
    }  
}
```

As you notice, actual action will be executed using the interceptor by **invocation.invoke()** call. So you can do some pre-processing and some post-processing based on your requirement.

The framework itself starts the process by making the first call to the `ActionInvocation` object's `invoke()`. Each time **invoke()** is called, `ActionInvocation` consults its state and executes whichever interceptor comes next. When all of the configured interceptors have been invoked, the `invoke()` method will cause the action itself to be executed. Following diagram shows the same concept through a request flow:



Create Action Class:

Let us create a java file HelloWorldAction.java under **Java Resources > src** with a package name **com.tutorialspoint.struts2** with the contents given below.

```
package com.tutorialspoint.struts2;

import com.opensymphony.xwork2.ActionSupport;

public class HelloWorldAction extends ActionSupport{
    private String name;

    public String execute() throws Exception {
        System.out.println("Inside action....");
        return "success";
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

This is a same class which we have seen in previous examples. We have standard getters and setter methods for the "name" property and an execute method that returns the string "success".

Create a View

Let us create the below jsp file **HelloWorld.jsp** in the WebContent folder in your eclipse project.

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
```

```
<title>Hello World</title>
</head>
<body>
  Hello World, <s:property value="name"/>
</body>
</html>
```

Create main page:

We also need to create **index.jsp** in the WebContent folder. This file will serve as the initial action URL where a user can click to tell the Struts 2 framework to call the a defined method of the HelloWorldAction class and render the HelloWorld.jsp view.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Hello World</title>
</head>
<body>
  <h1>Hello World From Struts2</h1>
  <form action="hello">
    <label for="name">Please enter your name</label><br/>
    <input type="text" name="name"/>
    <input type="submit" value="Say Hello"/>
  </form>
</body>
</html>
```

The **hello** action defined in the above view file will be mapped to the HelloWorldAction class and its execute method using struts.xml file.

Configuration Files

Now we need to register our interceptor and then call it as we had called default interceptor in previous example. To register a newly defined interceptor, the <interceptors>...</interceptors> tags are placed directly under the <package> tag ins **struts.xml** file. You can skip this step for a default interceptors as we did in our previous example. But here let us register and use it as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
  "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
  <constant name="struts.devMode" value="true" />
  <package name="helloworld" extends="struts-default">

    <interceptors>
      <interceptor name="myinterceptor"
        class="com.tutorialspoint.struts2.MyInterceptor" />
    </interceptors>

    <action name="hello"
      class="com.tutorialspoint.struts2.HelloWorldAction"
```

```
        method="execute">
        <interceptor-ref name="params"/>
        <interceptor-ref name="myinterceptor" />
        <result name="success">/HelloWorld.jsp</result>
    </action>

</package>
</struts>
```

It should be noted that you can register more than one interceptors inside **<package>** tag and same time you can call more than one interceptors inside the **<action>** tag. You can call same interceptor with the different actions.

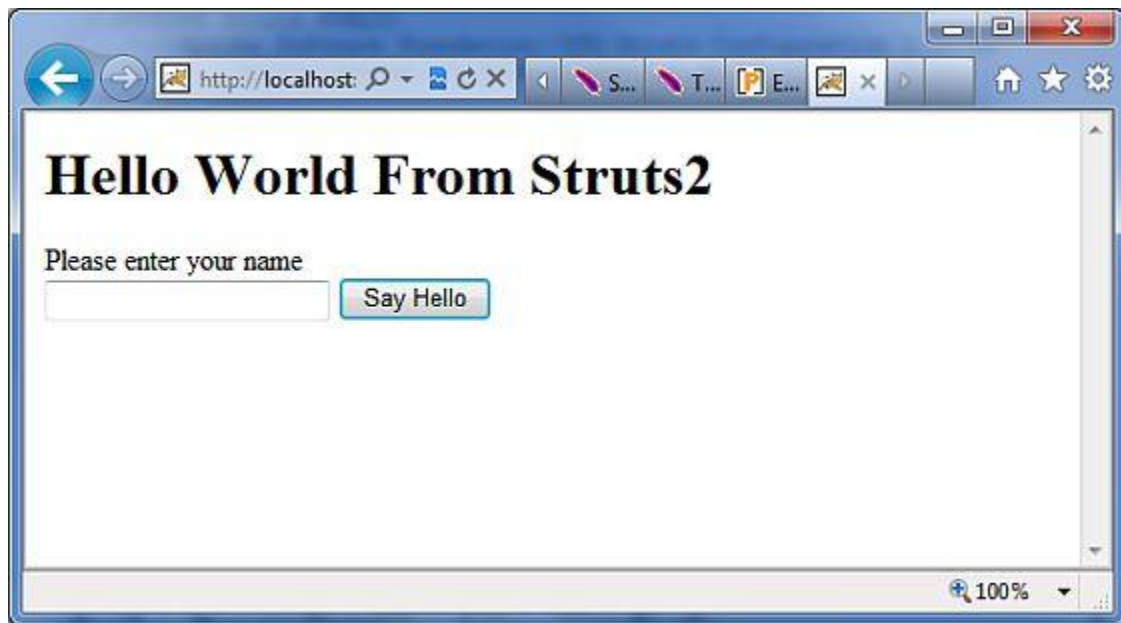
The web.xml file needs to be created under the WEB-INF folder under WebContent as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    id="WebApp_ID" version="3.0">

    <display-name>Struts 2</display-name>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>
            org.apache.struts2.dispatcher.FilterDispatcher
        </filter-class>
    </filter>

    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

Right click on the project name and click **Export > WAR File** to create a War file. Then deploy this WAR in the Tomcat's webapps directory. Finally, start Tomcat server and try to access URL <http://localhost:8080/HelloWorldStruts2/index.jsp>. This will give you following screen:



Now enter any word in the given text box and click Say Hello button to execute the defined action. Now if you will check the log generated, you will find following text at the bottom:

```
Pre-Processing
Inside action....
Post-Processing
```

Stacking multiple Interceptors:

As you can imagine, having to configure multiple interceptor for each action would quickly become extremely unmanageable. For this reason, interceptors are managed with interceptor stacks. Here is an example, directly from the struts-default.xml file:

```
<interceptor-stack name="basicStack">
  <interceptor-ref name="exception"/>
  <interceptor-ref name="servlet-config"/>
  <interceptor-ref name="prepare"/>
  <interceptor-ref name="checkbox"/>
  <interceptor-ref name="params"/>
  <interceptor-ref name="conversionError"/>
</interceptor-stack>
```

The above stack is called **basicStack** and can be used in your configuration as shown below. This configuration node is placed under the <package .../> node. Each <interceptor-ref .../> tag references either an interceptor or an interceptor stack that has been configured before the current interceptor stack. It is therefore very important to ensure that the name is unique across all interceptor and interceptor stack configurations when configuring the initial interceptors and interceptor stacks.

We have already seen how to apply interceptor to the action, applying interceptor stacks is no different. In fact, we use exactly the same tag:

```
<action name="hello" class="com.tutorialspoint.struts2.MyAction">
```

```
<interceptor-ref name="basicStack"/>
<result>view.jsp</result>
</action>
```

The above registration of "basicStack" will register complete stack of all the six interceptors with hello action. This should be noted that interceptors are executed in the order, in which they have been configured. For example, in above case, exception will be executed first, second would be servlet-config and so on.

Struts 2 Results and Result Types

As mentioned previously, the **<results>** tag plays the role of a **view** in the Struts2 MVC framework. The action is responsible for executing the business logic. The next step after executing the business logic is to display the view using the **<results>** tag.

Often there is some navigation rules attached with the results. For example, if the action method is to authenticate a user, there are three possible outcomes. (a) Successful Login (b) Unsuccessful Login - Incorrect username or password (c) Account Locked.

In this scenario, the action method will be configured with the three possible outcome strings and three different views to render the outcome. We have already seen this in the previous examples.

But, Struts2 does not tie you up with using JSP as the view technology. After all the whole purpose of the MVC paradigm is to keep the layers separate and highly configurable. For example, for a Web2.0 client, you may want to return XML or JSON as the output. In this case, you could create a new result type for XML or JSON and achieve this.

Struts comes with a number of predefined **result types** and whatever we've already seen that was the default result type **dispatcher**, which is used to dispatch to JSP pages. Struts allow you to use other markup languages for the view technology to present the results and popular choices include **Velocity**, **Freemaker**, **XSLT** and **Tiles**.

The dispatcher result type:

The **dispatcher** result type is the default type, and is used if no other result type is specified. It's used to forward to a servlet, JSP, HTML page, and so on, on the server. It uses the *RequestDispatcher.forward()* method.

We saw the "shorthand" version in our earlier examples, where we provided a JSP path as the body of the result tag.

```
<result name="success">
  /HelloWorld.jsp
</result>
```

We can also specify the JSP file using a **<param name="location">** tag within the **<result...>** element as follows:

```
<result name="success" type="dispatcher">
  <param name="location">
    /HelloWorld.jsp
  </param>
</result>
```

We can also supply a **parse** parameter, which is true by default. The parse parameter determines whether or not the location parameter will be parsed for OGNL expressions.

The FreeMaker result type:

In this example we are going to see how we can use **FreeMaker** as the view technology. Freemaker is a popular templating engine that is used to generate output using predefined templates. Let us create a Freemaker template file called **hello.fm** with the following contents:

```
Hello World ${name}
```

Here above file is a template where **name** is a parameter which will be passed from outside using the defined action. You will keep this file in your CLASSPATH. Next, let us modify the **struts.xml** to specify the result as follows:

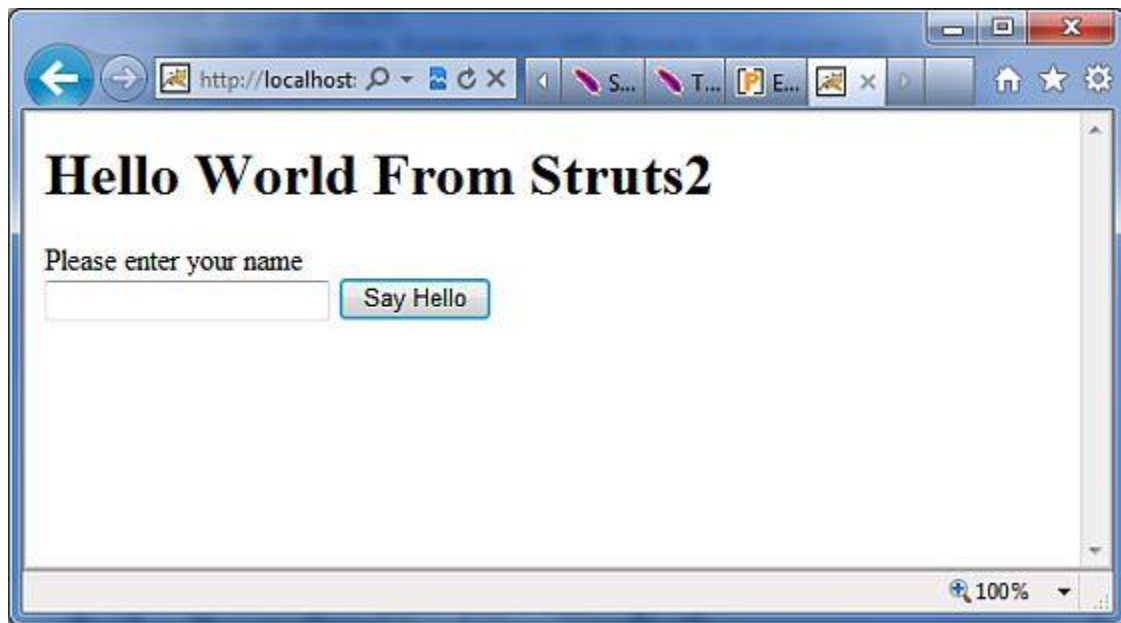
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
  <constant name="struts.devMode" value="true" />
  <package name="helloworld" extends="struts-default">

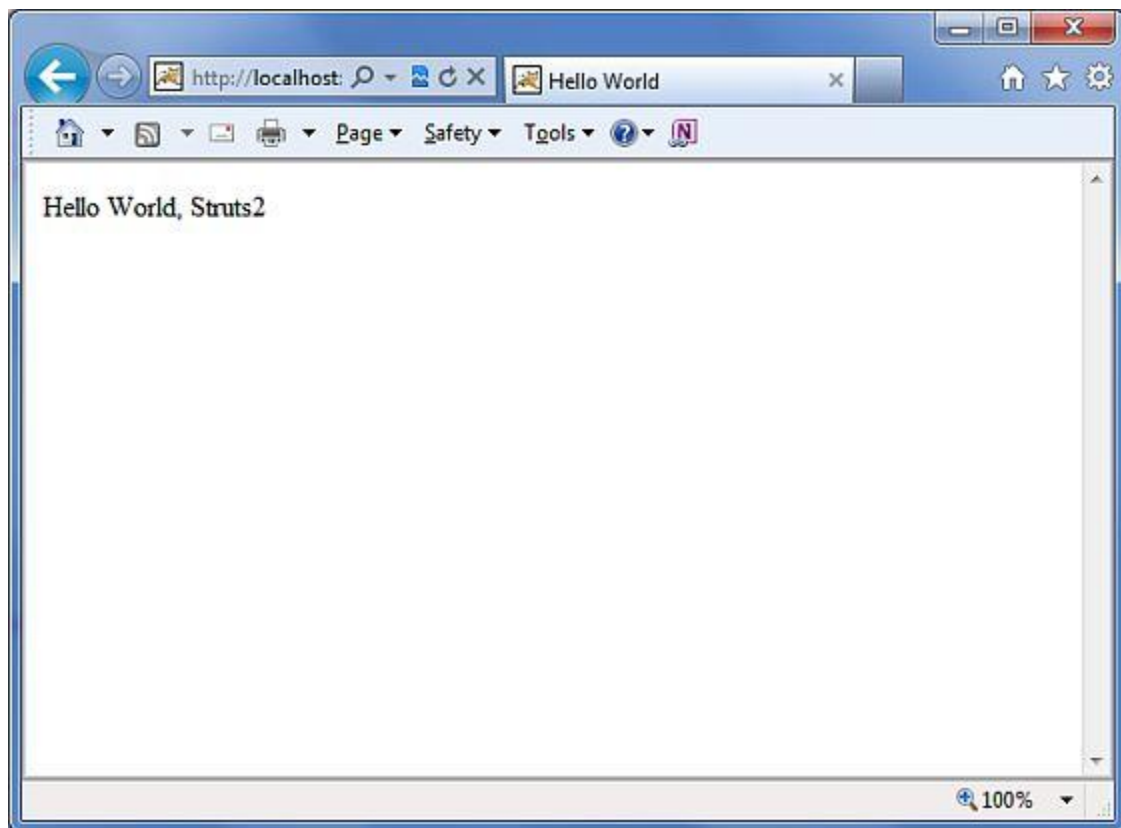
    <action name="hello"
      class="com.tutorialspoint.struts2.HelloWorldAction"
      method="execute">
      <result name="success" type="freemarker">
        <param name="location">/hello.fm</param>
      </result>
    </action>

  </package>
</struts>
```

Let us keep our HelloWorldAction.java, HelloWorldAction.jsp and index.jsp files as we have created them in examples chapter. Now Right click on the project name and click **Export > WAR File** to create a War file. Then deploy this WAR in the Tomcat's webapps directory. Finally, start Tomcat server and try to access URL <http://localhost:8080/HelloWorldStruts2/index.jsp>. This will give you following screen:



Enter a value "Struts2" and submit the page. You should see the next page



As you can see, this is exactly same as the JSP view except that we are not tied to using JSP as the view technology. We have used Freemaker in this example.

The redirect result type:

The **redirect** result type calls the standard `response.sendRedirect()` method, causing the browser to create a new request to the given location.

We can provide the location either in the body of the `<result...>` element or as a `<param name="location">` element. Redirect also supports the **parse** parameter. Here's an example configured using XML:

```
<action name="hello"
  class="com.tutorialspoint.struts2.HelloWorldAction"
  method="execute">
  <result name="success" type="redirect">
    <param name="location">
      /NewWorld.jsp
    </param >
  </result>
</action>
```

So just modify your `struts.xml` file to define redirect type as mentioned above and create a new file `NewWorld.jsp` where you will be redirected whenever hello action will return success.

Struts2 Value Stack/OGNL

The Value Stack:

The value stack is a set of several objects which keeps the following objects in the provided order:

SN	Objects & Description
1	Temporary Objects There are various temporary objects which are created during execution of a page. For example the current iteration value for a collection being looped over in a JSP tag.
2	The Model Object If you are using model objects in your struts application, the current model object is placed before the action on the value stack
3	The Action Object This will be the current action object which is being executed.
4	Named Objects These objects include <code>#application</code> , <code>#session</code> , <code>#request</code> , <code>#attr</code> and <code>#parameters</code> and refer to the corresponding servlet scopes

The value stack can be accessed via the tags provided for JSP, Velocity or Freemarker. There are various tags which we will study in separate chapters, are used to get and set struts 2.0 value stack. You can get valueStack object inside your action as follows:

```
ApplicationContext.getContext().getValueStack()
```

Once you have a ValueStack object, you can use following methods to manipulate that object:

SN	ValueStack Methods & Description
1	Object findValue(String expr) Find a value by evaluating the given expression against the stack in the default search order.
2	CompoundRoot getRoot() Get the CompoundRoot which holds the objects pushed onto the stack.
3	Object peek() Get the object on the top of the stack without changing the stack.
4	Object pop() Get the object on the top of the stack and remove it from the stack.
5	void push(Object o) Put this object onto the top of the stack.
6	void set(String key, Object o) Sets an object on the stack with the given key so it is retrievable by findValue(key,...)
7	void setDefaultType(Class defaultType) Sets the default type to convert to if no type is provided when getting a value.
8	void setValue(String expr, Object value) Attempts to set a property on a bean in the stack with the given expression using the default search order.
9	int size() Get the number of objects in the stack.

The OGNL:

The Object-Graph Navigation Language (OGNL) is a powerful expression language that is used to reference and manipulate data on the ValueStack. OGNL also helps in data transfer and type conversion.

The OGNL is very similar to the JSP Expression Language. OGNL is based on the idea of having a root or default object within the context. The properties of the default or root object can be referenced using the markup notation, which is the pound symbol.

As mentioned earlier, OGNL is based on a context and Struts builds an ActionContext map for use with OGNL. The ActionContext map consists of the following:

1. **application** - application scoped variables
2. **session** - session scoped variables
3. **root / value stack** - all your action variables are stored here
4. **request** - request scoped variables
5. **parameters** - request parameters
6. **attributes** - the attributes stored in page, request, session and application scope

It is important to understand that the Action object is always available in the value stack. So, therefore if your Action object has properties x and y there are readily available for you to use.

Objects in the ActionContext are referred using the pound symbol, however, the objects in the value stack can be directly referenced, for example if **employee** is a property of an action class then it can be referenced as follows:

```
<s:property value="name"/>
```

instead of

```
<s:property value="#name"/>
```

If you have an attribute in session called "login" you can retrieve it as follows:

```
<s:property value="#session.login"/>
```

OGNL also supports dealing with collections - namely Map, List and Set. For example to display a dropdown list of colors, you could do:

```
<s:select name="color" list="{ 'red', 'yellow', 'green' }" />
```

The OGNL expression is clever to interpret the "red", "yellow", "green" as colours and build a list based on that.

The OGNL expressions will be used extensively in the next chapters when we will study different tags. So rather than looking at them in isolation, let us look at it using some examples in the Form Tags / Control Tags / Data Tags and Ajax Tags section.

ValueStack/OGNL Example:

Create Action:

Let us consider following action class where we are accessing valueStack and then setting few keys which we will access using OGNL in our view ie. JSP page.

```
package com.tutorialspoint.struts2;

import java.util.*;

import com.opensymphony.xwork2.util.ValueStack;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionSupport;

public class HelloWorldAction extends ActionSupport{
    private String name;

    public String execute() throws Exception {
        ValueStack stack = ActionContext.getContext().getValueStack();
        Map<String, Object> context = new HashMap<String, Object>();

        context.put("key1", new String("This is key1"));
        context.put("key2", new String("This is key2"));
        stack.push(context);

        System.out.println("Size of the valueStack: " + stack.size());
    }
}
```

```
        return "success";
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Actually, Struts 2 adds your action to the top of the valueStack when executed. So, the usual way to put stuff on the Value Stack is to add getters/setters for the values to your Action class and then use `<s:property>` tag to access the values. But I'm showing you how exactly ActionContext and ValueStack work in struts.

Create Views

Let us create the below jsp file **HelloWorld.jsp** in the WebContent folder in your eclipse project. This view will be displayed in case action returns success:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Hello World</title>
</head>
<body>
    Entered value : <s:property value="name"/><br/>
    Value of key 1 : <s:property value="key1" /><br/>
    Value of key 2 : <s:property value="key2" /> <br/>
</body>
</html>
```

We also need to create **index.jsp** in the WebContent folder whose content is as follows:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Hello World</title>
</head>
<body>
    <h1>Hello World From Struts2</h1>
    <form action="hello">
        <label for="name">Please enter your name</label><br/>
        <input type="text" name="name"/>
        <input type="submit" value="Say Hello"/>
    </form>
</body>
</html>
```

Configuration Files

Following is the content of **struts.xml** file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
    <constant name="struts.devMode" value="true" />
    <package name="helloworld" extends="struts-default">

        <action name="hello"
            class="com.tutorialspoint.struts2.HelloWorldAction"
            method="execute">
            <result name="success">/HelloWorld.jsp</result>
        </action>

    </package>
</struts>
```

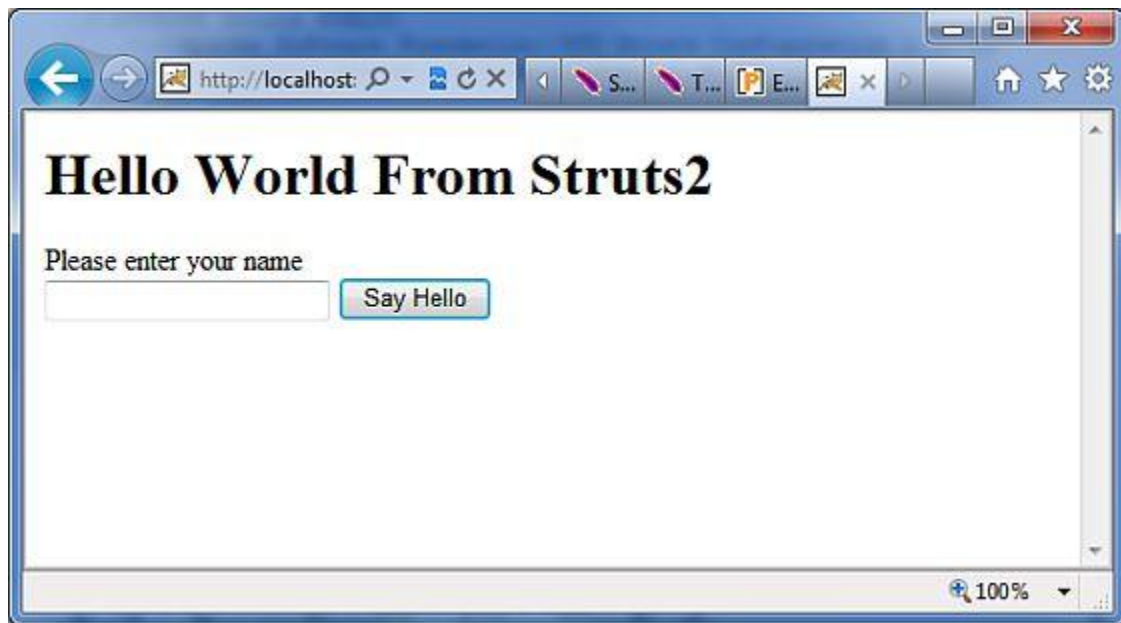
Following is the content of **web.xml** file:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    id="WebApp_ID" version="3.0">

    <display-name>Struts 2</display-name>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>
            org.apache.struts2.dispatcher.FilterDispatcher
        </filter-class>
    </filter>

    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

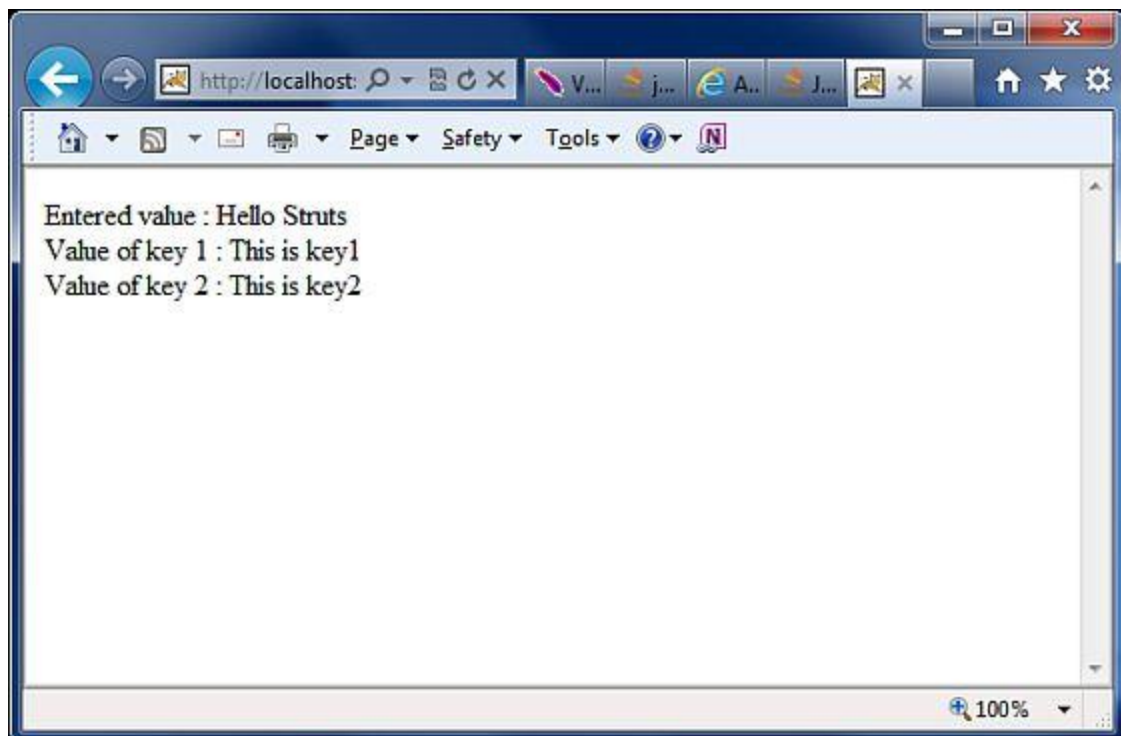
Right click on the project name and click **Export > WAR File** to create a War file. Then deploy this WAR in the Tomcat's webapps directory. Finally, start Tomcat server and try to access URL <http://localhost:8080/HelloWorldStruts2/index.jsp>. This will give you following screen:



Now enter any word in the given text box and click "Say Hello" button to execute the defined action. Now if you will check the log generated, you will find following text at the bottom:

```
Size of the valueStack: 3
```

and this will display following screen, which will display whatever value you will enter and value of key1 and key2 which we had put on ValueStack.



Conclusion:



Tutorials Point, Simply Easy Learning

Hope you have good grip on Struts 2 concepts. If you are willing to know more detail and further examples like file upload, sending email, form validations etc. then you can go through the complete tutorial.

Further Detail:

Refer to the link http://www.tutorialspoint.com/struts_2

List of Tutorials from TutorialsPoint.com	
<ul style="list-style-type: none">▪ Learn JSP▪ Learn Servlets▪ Learn log4j▪ Learn iBATIS▪ Learn Java▪ Learn JDBC▪ Java Examples▪ Learn Best Practices▪ Learn Python▪ Learn Ruby▪ Learn Ruby on Rails▪ Learn SQL▪ Learn MySQL▪ Learn AJAX▪ Learn C Programming▪ Learn C++ Programming▪ Learn CGI with PERL▪ Learn DLL▪ Learn ebXML▪ Learn Euphoria▪ Learn GDB Debugger▪ Learn Makefile▪ Learn Parrot▪ Learn Perl Script▪ Learn PHP Script▪ Learn Six Sigma▪ Learn SEI CMMI▪ Learn WiMAX▪ Learn Telecom Billing	<ul style="list-style-type: none">▪ Learn ASP.Net▪ Learn HTML▪ Learn HTML5▪ Learn XHTML▪ Learn CSS▪ Learn HTTP▪ Learn JavaScript▪ Learn jQuery▪ Learn Prototype▪ Learn script.aculo.us▪ Web Developer's Guide▪ Learn RADIUS▪ Learn RSS▪ Learn SEO Techniques▪ Learn SOAP▪ Learn UDDI▪ Learn Unix Sockets▪ Learn Web Services▪ Learn XML-RPC▪ Learn UML▪ Learn UNIX▪ Learn WSDL▪ Learn i-Mode▪ Learn GPRS▪ Learn GSM▪ Learn WAP▪ Learn WML▪ Learn Wi-Fi▪ Learn Apache ANT
webmaster@TutorialsPoint.com	

